Technical Report UW-CSE-17-08-02

Porting Hyperkernel to the ARM Architecture

Dylan Johnson
University of Washington
dgj16@cs.washington.edu

Keywords ARM, AArch64, Exokernel, Operating Systems, Virtualization

Abstract

This work describes the porting of Hyperkernel, an x86 kernel, to the ARMv8-A architecture. Hyperkernel was created to demonstrate various OS design decisions that are amenable to push-button verification. Hyperkernel simplifies reasoning about virtual memory by separating the kernel and user address spaces. In addition, Hyperkernel adopts an exokernel design to minimize code complexity, and thus its required proof burden. Both of Hyperkernel's design choices are accomplished through the use of x86 virtualization support. After developing an x86 prototype, advantageous design differences between x86 and ARM motivated us to port Hyperkernel to the ARMv8-A architecture. We explored these differences and benchmarked aspects of the new interface Hyperkernel provides on ARM to demonstrate that the ARM version of Hyperkernel should be explored further in the future. We also outline the ARMv8-A architecture and the various design challenges overcome to fit Hyperkernel within the ARM programming model.

1. Introduction

Hyperkernel was created to demonstrate the techniques for building an OS that can be formally verified using push-button techniques, such as satisfiability modulo theories (SMT). The design of Hyperkernel emphasized a small, statically allocated, and simple exokernel. During development, it became clear that placing Hyperkernel in non-root ring 0 would lead to problems. First, the virtual memory model on modern operating systems cannot be verified in a reasonable amount of time using SMT solvers, as the kernel and user address space are shared. Second, an exokernel built using the typical x86 programming model would have trouble exporting hardware control to user space applications. To solve these issues, Hyperkernel was designed to run within x86's root CPU mode. While traditionally reserved for hypervisors, root mode created a clean separation between the kernel and user address spaces, making it easier to prove properties of isolation. In addition, a kernel running in root mode can be greatly simplified by exporting hardware resource management to user processes in non-root ring 0.

This paper provides a single core contribution. We present an investigation into the ability of the ARM architecture to provide an appealing foundation for push-button verified operating systems. When porting Hyperkernel to ARM we emphasized maintaining the original features of Hyperkernel that allowed it to be easily verified on x86. Additionally, the ARM architecture created the possibility of optimizing parts of the Hyperkernel implementation. For example, ARM provides greater flexibility to developers during transitions into hypervisor mode, indicating that virtual machine abstractions not maintained in Hyperkernel could lead to faster mode transitions on ARM. We outline these optimizations and benchmark a few of the design decisions made during the porting of Hyperkernel to ARM.

The rest of this paper is organized as follows. §2 discusses work similar to porting Hyperkernel to ARM. §3 provides an overview of the ARM architecture. §4 details the design of Hyperkernel on ARM. §5 describes the implementation of Hyperkernel on ARM and the differences between the x86 and ARM versions. §6 presents our experiments and the performance of ARM hypercalls. §7 concludes.

2. Related work

OS design. Hyperkernel contains several similarities to Dune, where each process is virtualized, allowing them to have more direct control over hardware features such as interrupt vectors [3]. Dune uses a small

Linux kernel module to mediate several running virtualized processes, and a user-level library aids in managing the newly-exposed resources.

Hyperkernel also draws inspiration from Exokernel [6]. An exokernel is an operating system kernel that provides user level management of operating system abstractions. In an exokernel, user applications are allowed to manage hardware resources, while the kernel enforces resource policies for the entire system. This design provides a platform for domain specific optimizations, increased flexibility for application builders, and better optimizations for specific workloads [6]. Previous exokernel implementations such as Aegis [6] and Xok [7] provided low-level abstractions by running user code in the same address space and CPU mode as privileged kernel code. This allowed the sharing of memory, but fault isolation was difficult to guarantee. In Hyperkernel, low-level abstractions are provided by leveraging virtual machine support. These extensions were previously thought to be too slow to be efficient [6] but recent popularity of hardware virtualization support has slowly improved the performance and in some cases the costs have been mitigated entirely [4].

Hypervisor design. KVM-ARM is a hypervisor that was built and accepted into the mainline Linux kernel. Similar to Hyperkernel, KVM-ARM utilizes the ARM programming model in an uncoventional way to both reduce and simplify the code required. Specifically, it uses a new hypervisor design called split-mode virtualization, where only a small portion of hypervisor execution uses the ARM virtualization extensions. Instead, the KVM-ARM hypervisor transitions back into the host kernel where it can take full advantage of already built OS mechanisms [5].

3. ARM Architecture Overview

The ARMv8-A architecture, which we will refer to as simply the ARM architecture from here forward, was designed for full application workloads on the Cortex-A family cores. It defines a rich programming model for applications, operating systems, and hypervisors. This section summarizes the core pieces of the ARM architecture and how they were utilized in the implementation of Hyperkernel on ARM.

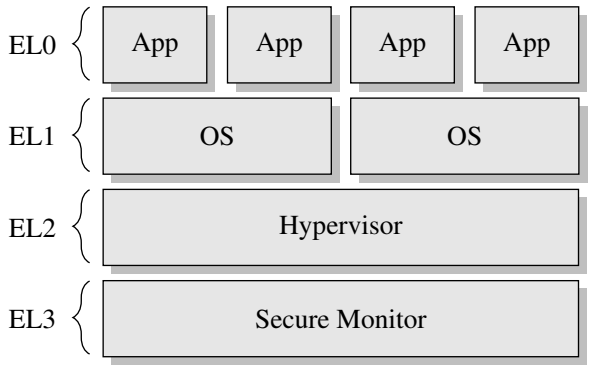

Figure 1. The ARM programming model. Moving down the exception levels increases privilege level. The secure exceptions levels are omitted for clarity.

3.1 Exception levels

Figure 1 shows the four possible CPU modes defined by the ARM architecture. Each mode is referred to as an exception level (abbreviated EL0-EL3) that determines the privilege of currently executing software. Transitions into higher exception levels are caused by either synchronous or asynchronous events, called exceptions. An exception can only be taken to the next highest exception level or to the current exception level when above EL0. When returning from an exception, the exception level can only decrease by a single level or stay the same [1]. In addition, each exception level supports two execution states, AArch32 and AArch64. AArch32 uses 32-bit widths and is backwards compatible with the ARMv7-A architecture while AArch64 uses 64-bit widths and is a new feature introduced in ARMv8. Hyperkernel currently supports only AArch64 in order to simplify its implementation.

ARM designed each exception level to provide specific capabilities for certain types of system level software. For example, the least privileged level, EL0, typically runs user applications. EL0 has limited exposure to system configuration registers and exception handling in order to provide process isolation. EL1 is designed for operating system execution. It exposes exception handling, page table modification, and many other system configuration options. EL2 equips software with hardware support for virtualization, which is typically utilized by a hypervisor. Software in EL2 is given the tools to manage virtual memory, exception handling, timers, and other sensitive system components to provide isolation between virtual machines in EL1. EL2 has the ability to trap virtual machines into

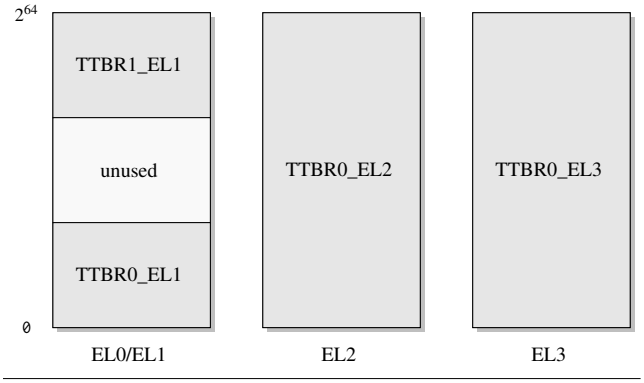

Figure 2. The ARM virtual memory model. EL0 and EL1 share an address space across two page table base registers, TTBR0_EL1 and TTBR1_EL1. EL2 and EL3 have separate address spaces.

EL2, where it can emulate or forward sensitive operations. The last and most privileged level is EL3, which has the ability to switch between a non-secure and secure execution state. These two states are orthogonal to the exception levels and secure mode grants access to a separate physical memory. Software running in EL3 typically manages the transitions between the secure and non-secure CPU modes.

3.2 Virtual Memory Model

Figure 2 shows a graphical representation of the ARM virtual memory model. Each exception level above EL0 has control over the configuration of the MMU when executing in their respective levels. For page table managment, ARM banks page table base registers (called translation table base registers, TTBRs) across each exception level. In contrast, an x86 CPU has a single page table base register for all CPU modes. EL1 and EL0 share an address space accross two TTBRs, TTBR0_EL1 and TTBR1_EL1, that are under EL1 control. This shared address space is split in half with TTBR1_EL1 for the upper addresses and TTBR0_EL1 for the lower addresses. This split provides a convenient mechanism for splitting a kernel and user address space.

Memory virtualization is provided by a second stage of address translation similar to the extended page tables (EPT) on x86. A virtual machine running in EL1 will first translate virtual addresses into intermediate physical addresses (IPA) using its own page table. Next, the second stage translates IPAs into physical addresses using a page table provided by the hypervisor in EL2. A hypervisor can use the second stage of translation to protect MMIO peripherals from direct control by a VM by forcing data abort exceptions that route to EL2.

3.3 TLB

The ARM virtual memory model provides features for operating system developers to mitigate unnecesary TLB flushes. For example, ARM TLB's can match entries based on the current exception level, removing the need to flush the TLB when taking an exception [1]. Similarly on x86, a vm exit into root mode will not cause a TLB flush. In addition, ARM defines a unique application specific identifier (ASID) that an OS assigns to each user process. ASID's are communicated to the hardware through the top sixteen bits of TTBR0_EL1. During an address translation, the hardware matches TLB entries using the ASID, removing the need to flush the TLB during a context switch. On x86, a similar mechansim is provided called the process-context identifier (PCID).

3.4 Interrupts

Interrupts on ARM are managed and routed by ARM's Generic Interrupt Controller (GIC). The GIC architecture defines mechansims to control the generation, routing, and priority of interrupts in a system [2]. The third generation of the GIC architecture, GICv3, defines two compontents of a GIC, the distributor and redistributors. Each system has a single distributor, which handles global interrupts and interprocessor interrupts, while each CPU has a redistributor, which manages local interrupts (e.g. caused by local timers). Interaction with the GIC distributor is provided through an MMIO interface. Interaction with the GIC redistributors is provided through a set of configuration registers.

The GICv3 architecture defines a virtual GIC (vGIC) interface for each CPU that provides virtual interrupt management capabilities to a hypervisor. When virtual interrupts are enabled, all physical interrupts are routed to EL2. Before the target VM is scheduled on a physical CPU, the hypervisor signals for a virtual interrupt to be delivered once execution has returned to EL1. If a VM attempts to configure the GIC distributor through the MMIO interface, the second stage of translation will trap execution into EL2 where the hypervisor can emulate the operation. VM accesses to the GIC system registers are routed to a set of virtual system registers, which control virtual interrupt delivery. During a virtual CPU context switch, the hypervisor must save the

writable virtual system register state, the state of any pending virtual interrupts, and the VM specific virtual interrupt configuration.

3.5 Timers

Each ARM CPU contains two counters, a physical counter and a virtual counter that indicate the passage of time. The virtual counter value is calculated by subtracting an offset from the physical counter. These two counters power a set of generic timers accessible from each CPU mode. The physical counter powers an EL1 timer and an EL2 timer, while the virtual counter powers a virtual timer. All three timers support upcounter and downcounter interrupts that will only generate an interrupt for its assigned CPU. Traditionally, a hypervisor will use the EL1 timer to preempt virtual machine execution, while virtual machines in EL1 use the virtual timer to preempt user processes. To avoid virtual machines from observing the physical passage of time, any accesses to the EL1 timer from EL1 are trapped into EL2.

4. ARM Hyperkernel Design

4.1 Exception Levels

Similar to Hyperkernel on x86, the ARM version of Hyperkernel required that the kernel and user address spaces be clearly separated. In addition, the ARM version of Hyperkernel needed the ability to easily export low level hardware resources to user space, such as page table and exception management. The placement of Hyperkernel within the correct exception level was crucial for providing these features. This section describes in detail the factors considered when placing Hyperkernel within the ARM programming model.

We had three options for the placement of Hyperkernel user processes, EL0, EL1, and EL2. EL0 does not have access to many system configuration registers, making it too restricted for an exokernel to properly provide low-level hardware control. Placing user processes in EL2 also creates various difficulties. First, Hyperkernel would not be able to assign ASIDs to processes running in EL2, which would result in a TLB flush during every context switch. Second, user processes in EL2 would have full control over TTBR0_EL2. Hyperkernel would be protected by secure memory in EL3, but a malicious process could break isolation by manually changing its page tables to expose another processes physical memory. Third, because virtual interrupts can only be routed to EL1, exporting interrupt handling to EL2 would require software to register interrupt handler upcalls to the kernel, which complicates the features already provided by the vGIC. Thus the most beneficial choice was to place user space in EL1.

With user processes in EL1, Hyperkernel was restricted to either EL2 or EL3. Placing Hyperkernel in EL3 provides the same functionality as placing Hyperkernel in EL2, because EL3 is strictly more privileged than EL2, but we decided to place Hyperkernel in EL2 because features such as the security modes would unnecessarily complicate the implementation of Hyperkernel. Ultimately, Hyperkernel was placed in EL2 and user processes were placed in EL1.

4.2 Virtual Memory Model

Initially, Hyperkernel was designed to execute in EL2 with the MMU disabled, simplifying Hyperkernel even further by removing the need for kernel page tables. In addition, when the MMU is disabled, isolation guarantees between the kernel and user space become significantly easier to verify. Unfortunately, disabling the MMU on ARM has the side effect of permanently disabling the data cache while in EL2. Our experiments, detailed in §6, analyze the performance impact of disabling the MMU while in EL2. Consequently, Hyperkernel was adapted to execute within EL2's address space, which is separate from EL1, EL0 and EL3.

In Hyperkernel, user space exclusively uses TTBR0_EL1, which provides a maximum of 2^{48} virtual addresses. Any accesses to virtual memory that are above the TTBR0_EL1 region trap into Hyperkernel. User processes can modify their page tables by using a set of hypercalls provided by Hyperkernel. Any direct write of TTBR0_EL1 is trapped into Hyperkernel to prevent user processes from violating isolation between both the kernel and other processes.

Hyperkernel user processes are aware that they do not have access to portions of their address space. In contrast, a virtual machine expects full control over all virtual addresses. Typically a hypervisor provides this abstraction through stage 2 translations. Hyperkernel disables stage 2 address translation altogether, since it controls EL1 virtual memory through page mapping syscalls. Some hypervisor virtualization duties rely on stage 2 translation, such as emulation of the GIC MMIO interface and access to most I/O devices.

Hyperkernel restricts access to the these regions of memory by refusing to map any virtual pages on top of these MMIO interfaces.

4.3 TLB

Since ARM TLBs support entry matching using the current exception level, a transition from user space into the kernel does not require a TLB flush. In addition, Hyperkernel takes advantage of ASIDs by managing an ASID for every process. Although ASIDs were originally designed to identify user processes in EL0, they can also be used to identify user processes in Hyperkernel because they are forced to use TTBR0_EL1 exclusively. When a specific process is scheduled on a CPU, TTBR0_EL1 is updated with the correct ASID.

4.4 Interrupts

Hyperkernel configures the system to route all physical interrupts to EL2. When Hyperkernel is notified of an interrupt, it determines if the interrupt should be signaled to a user process in EL1. If the final destination is a process in EL1, then it adds the interrupt to a list of pending interrupts for that process, otherwise Hyperkernel handles the interrupt directly. Just before a process is scheduled its pending interrupts are forwarded to the vGIC, which generates virtual interrupts when the processor returns to EL1. Handling a virtual interrupt in a Hyperkernel user process is no different than handling an interrupt in a typical OS.

4.5 Timers

A user processes view of physical time can advance significantly between the execution of two instructions. In contrast, a virtual machine requires timers that will only advance when one of its vCPUs is scheduled. Hyperkernel takes advantage of this by removing the virtual timer logic, which saves on the cost of transitions into EL2. In addition, user processes are given full control of the EL1 timer, which removes the timer emulation logic required in Hyperkernel. The EL2 timer is used by the Hyperkernel scheduler to preempt processes.

5. Implementation

This section outlines the effort of porting Hyperkernel from x86 to ARM. During the port, we focused on reusing many of the mechanisms already present in the original version of Hyperkernel. Fortunately, some aspects of the x86 and ARM architecture share similarities, allowing significant code reuse between versions.

Process State	
State (Zombie, etc)	Kernel Stack
Saved CPU State	Process ID
Open File Descriptors	Pending Interrupts
IPC Information	

Figure 3. The information stored for every process in Hyperkernel on both x86 and ARM.

Page table management. The x86 page table management code was significantly reused for the ARM version of Hyperkernel. This is because both versions of Hyperkernel use 4KB pages, 48 bit virtual addresses, and 4 levels of virtual address translation. Functions that walked the page tables, inserted new page table entries, or allocated new pages only needed small changes in order to work on ARM. The largest change to these functions was factoring out the formatting and setting of permissions within page table entries. Fortunately, the permissions for both descriptor formats were similar and conversion from one format to another merely required rearranging the permissions.

Process management. Figure 3 summarizes the information required to maintain a process in Hyperkernel on both x86 and ARM. The field that differs substantially between x86 and ARM is the saved CPU state. On x86, the saved CPU state is stored within a virtual machine control structure (VMCS), while on ARM the saved CPU state is simply a struct that holds the information detailed in Figure 4. During a context switch, hypercall, or trap into EL2, software copies the required state into the saved CPU state struct. During a return into EL1, this state is copied back to the CPU.

Drivers and components. Many Hyperkernel functionalities required new drivers and components to communicate with ARM hardware. For example, SMP multicore management operations such as enabling a CPU core are achieved through a power management system called the power state coordination interface (PSCI). A system with EL3 implemented exposes a PSCI conduit through synchronous exceptions into EL3. Hyperkernel generates PSCI calls to EL3, which is expected to perform the necessary MMIO communication to enable or disable CPU cores. Other components implemented include a PL011 UART driver

Action	NR	State Saved
Context Switch	31	GP Integer Registers (x0-x30)
	4	EL1 and EL0 PC/SP
	5	EL1 Exception Information
	6	EL1 MMU Information
	1	EL1 System Configuration
	1	EL1 Timer Configuration
	4	EL1 and EL0 Thread Identification
	15	GIC State
Hypercall	31	GP Integer Registers (x0-x30)
	4	EL1 and EL0 PC/SP

Figure 4. The state saved during either a context switch or hypercall in Hyperkernel. The number of register reads/writes in each group is shown in the NR column.

for serial communication, a simple timer module for scheduling, and a GICv3 driver for interrupt management.

5.1 Development Effort

The porting of the Hyperkernel core and a basic init process took 5 person-months and added 3197 SLOC to Hyperkernel. Much of the time and effort porting Hyperkernel to ARM was spent studying the x86 and ARM architectures and their various peripheral devices. Currently Hyperkernel on ARM is missing a complete implementation of user space, which is left as future work.

5.2 Hyperkernel on ARM Hardware

During development we utilized QEMU, a full system hardware emulator, to test the execution of Hyperkernel on a virtual ARM system. A new bootloader and small modifications to Hyperkernel were required in order to run Hyperkernel on real hardware. This section details the process of running Hyperkernel on a HiSilicon HiKey board that contains a Cortex A53 CPU.

QEMU to HiKey The virtual ARM boards emulated by QEMU provided an accurate representation of real ARM hardware. The only modification to Hyperkernel was a change to the base address of DRAM and where Hyperkernel was loaded into physical memory during bootup. This was required because the QEMU virtual board placed DRAM at a different starting address than the HiKey board.

Bootloader Hyperkernel spoofs itself as a Linux kernel image in order for QEMU to recognize and load Hyperkernel into memory during bootup. When running Hyperkernel on real hardware, we required a compatible bootloader to perform a similar operation. Unfortunately, the current ARM version of the GRUB bootloader only supports Linux kernels that contain an EFI stub. Instead of adding an EFI stub to Hyperkernel, we decided to create a custom EFI bootloader using the Linaro EFI developer toolchain. This custom bootloader simply copies Hyperkernel into the start of DRAM, loads a device tree blob adjacent to it, and jumps to the beginning of Hyperkernel.

6. Experiments and Results

The current state of Hyperkernel on ARM includes the kernel and a simple implementation of the init process. In order to run a full benchmark to compare the ARM and x86 versions of Hyperkernel a complete user space implementation is required. Instead of creating a full benchmark that measures the impact of porting Hyperkernel to ARM, we decided to measure the side effects of a few design decisions of Hyperkernel on ARM using a microbenchmark.

A typical ARM operating system interfaces with user proceses through system calls generated by an `svc` instruction. Since Hyperkernel runs in EL2 instead of EL1, a user process must use an `hvc` instruction (hypercall) to request a more privileged operation from Hyperkernel. This section analyzes the performance implications of using hypercalls in Hyperkernel instead of system calls. In addition, we evaluate the performance impacts of enabling and disabling the MMU, data cache, and instruction cache to determine if execution in EL2 is viable when the MMU is disabled.

We performed our experiments with a microbenchmark adapted from Dune [3] that measures the number of cycles required to execute a system call (`svc`/`eret` pair), and a hypercall `hvc`/`eret` pair). This microbenchmark executed as the init process on an early version of Hyperkernel on the HiKey Cortex-A53 CPU. The results are shown in Figure 5: when all three features are enabled, a hypercall and system call require identical amounts of cycles. It is plausible that the `svc` and `hvc` pathways are reused within the microarchitecture, because exception routing is a common operation. In addition, this result indicates that the performance differences measured between hypercalls and system calls in past work is caused by the saving of extra state only.

MMU	D Cache	I Cache	syscall	hypercall
Disabled	Disabled	Disabled	2200	904
Disabled	Disabled	Enabled	568	48
Enabled	Disabled	Disabled	1727	658
Enabled	Disabled	Enabled	40	36
Enabled	Enabled	Disabled	1695	659
Enabled	Enabled	Enabled	36	36

Figure 5. Cycle counts of system calls and hypercalls on an ARM Cortex A53. Each result averages 50 million trials.

The first iteration of Hyperkernel on ARM disabled the MMU when in EL2 in an effort to remove the in kernel virtual memory management. Unfortunately, when the ARM MMU is disabled, the data cache is permanently disabled because the cacheability of memory is stored within page table entries. The same is not true of the instruction cache, which can still be enabled when the MMU is disabled. Our results demonstrate that the instruction cache has a significant effect on the performance of both a hypercalls and syscalls. The lack of performance impact when disabling the data cache can be explained by the absense of data accesses between the pairs of instructions benchmarked. When data accesses are inserted between the pairs, the number of cycles increases in magnitude similar to when the instruction cache is disabled. After these measurements we redesigned Hyperkernel to enable the MMU after the creation of a simple identity mapping.

Recent work has shown that a hypercall on x86 can take upwards of 800 cycles on recent microarchitectures; much more expensive than a 70 cycle syscall [4]. Our results indicate that a hypercall on ARM could be consistently faster than a hypercall on x86 if enough optimizations are applied to the transitions into EL2. Measuring the cost of a full hypercall in Hyperkernel is planned as future work.

7. Conclusion

This paper presented the porting of Hyperkernel, an x86 kernel, to the ARMv8-A architecture and the various design challenges faced when shaping it for the ARM programming model. We presented core aspects of the ARM architecture and how they were utilized within Hyperkernel. Our experience shows that the ARM version of Hyperkernel opens the possibility of optimizing both the transitions into Hyperkernel and the abstractions it provides to user processes. We believe that ARM provides a solid foundation for developers to create operating systems that are verified using push-button verification.

References

[1] ARM. *ARM Architecture Reference Manual ARMv8-A DDI0487A*, 2015.

[2] ARM. *ARM Generic Interrupt Controller Architecture Specification IHI0069C*, 2016.

[3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, Hollywood, CA, Oct. 2012.

[4] E. Bugnion, J. Nieh, and D. Tsafrir. *Hardware and software support for virtualization*. Morgan & Claypool, 2017.

[5] C. Dall and J. Nieh. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 333–347, Salt Lake City, UT, Mar. 2014.

[6] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[7] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, Saint-Malo, France, Oct. 1997.